Programação concorrente

10 – Threads

LEEC

João Nuno Silva

TÉCNICO LISBOA

Bibliography

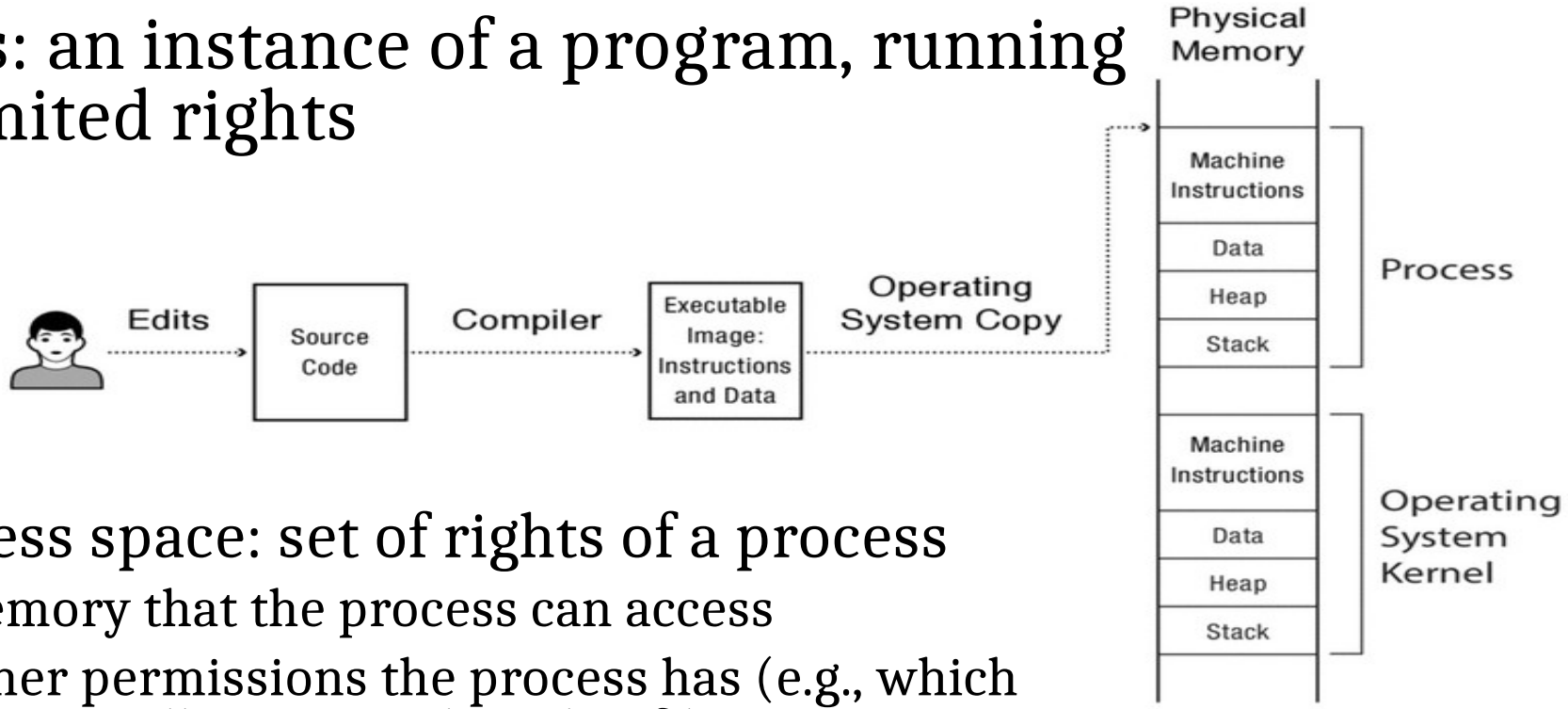
- Operating Systems Principles & Practice Volume II: Concurrency. Thomas Anderson, Mike Dahlin.
  - Chapter 4
- The Linux Programming Interface, Michael Kerrisk,
  - Chapter 29
- Multithreaded Programming Guide, Sun
  - Chapters 1 and 2

Parallelism/concurrency

- OS need to support concurrency
  - Multiple tasks running at the same time
- Simplest abstractions
  - Process
  - Thread
- On multiprocessors
  - Processes and threads allow parallelism

Process Abstraction

- Process: an instance of a program, running with limited rights



  - Address space: set of rights of a process
    - Memory that the process can access
    - Other permissions the process has (e.g., which system calls it can make, what files it can access

Processes

- Distributed-memory multi-computers
  - Process fits perfectly
  - Independent memory nodes
    - Independent memory for processes
- Parallel programming models
  - Tasks and Channels
  - Message passing
  - Data Parallelism

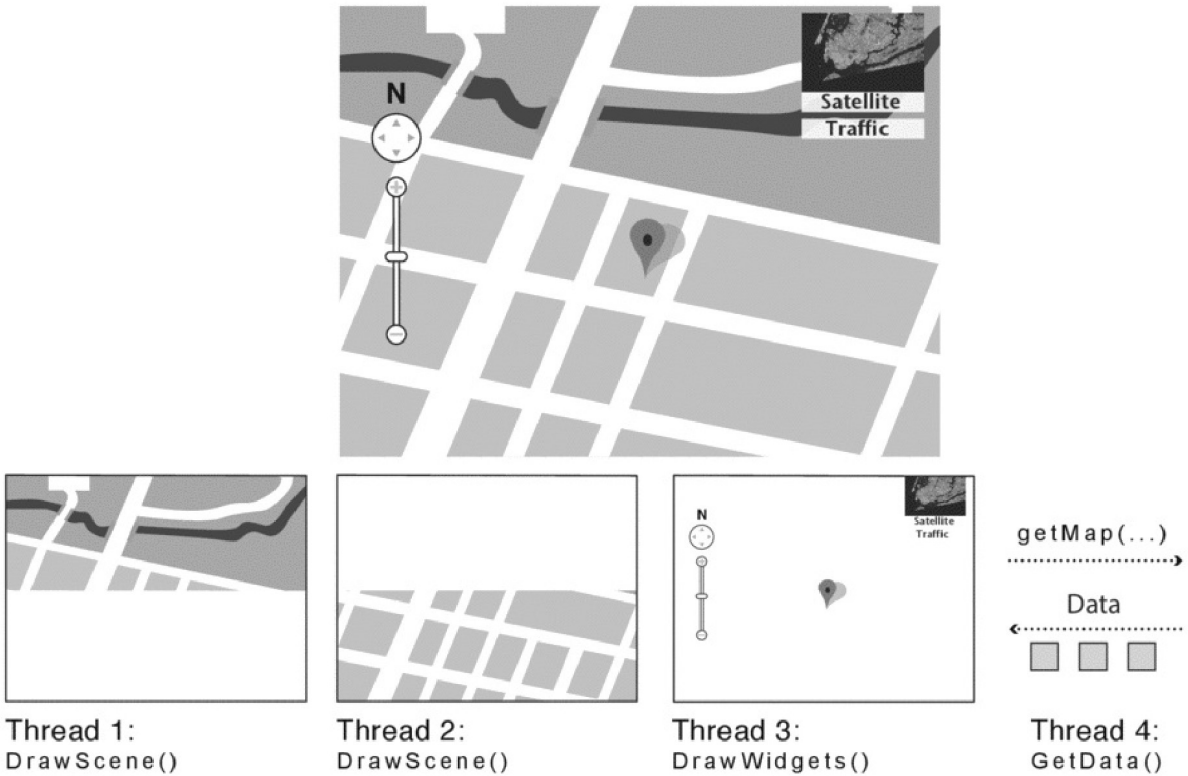
Processes

- Shared-memory multi-computers
  - Processes are isolated entities
  - Shared memory is complex
- Requires Middle-ware
  - Launch of processes
  - Explicit communication

Threads

- A Thread is an independent stream of instructions that can be schedule to run as such by the OS
  - Concurrent procedure inside a process
  - Runs independently from its main program.
- A Thread exists within a process and uses the process resources.

Thread

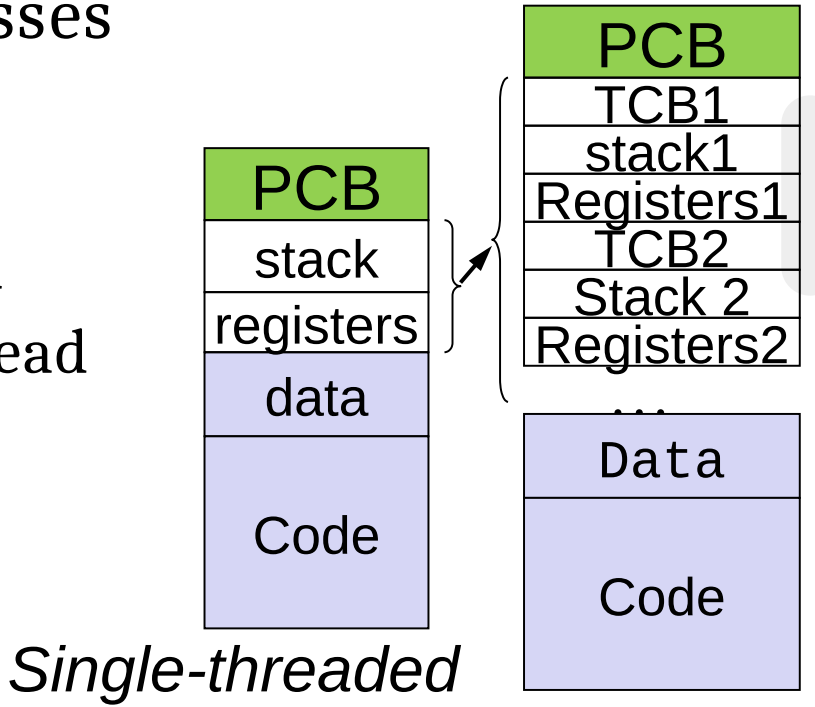

N
Satellite
Traffic
getMap(...)
Data
Thread 1:
DrawScene()
Thread 2:
DrawScene()
Thread 3:
DrawWidgets()
Thread 4:
GetData()

Threads resources

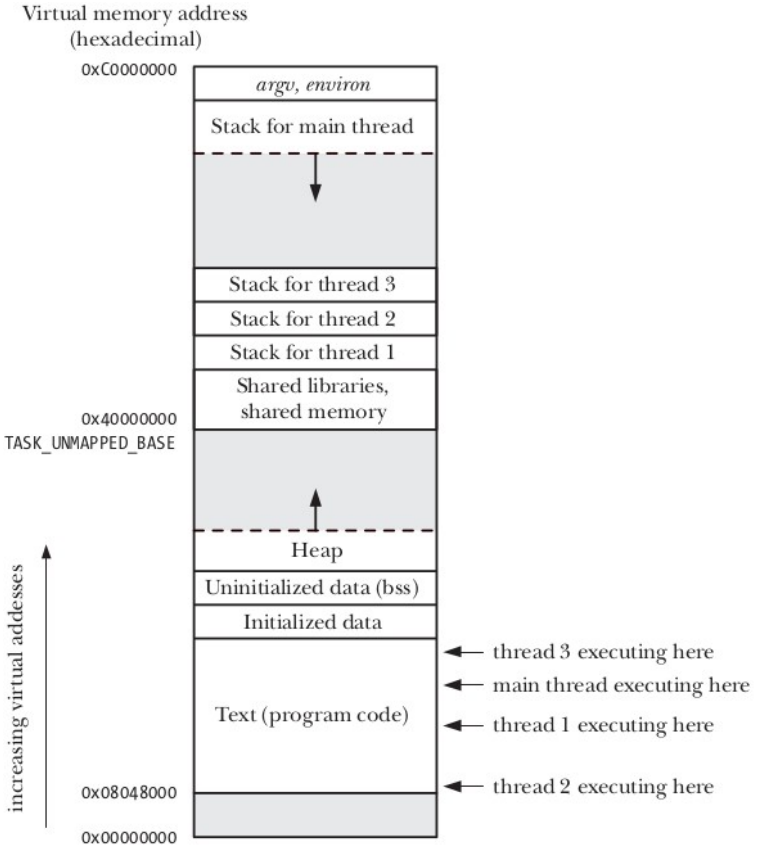
- Threads only duplicate the essential resources
  - necessary to be independently scheduled
- A thread will die if the parent process dies.
- A thread is “lightweight”
  - most of the overhead has already been accomplished through the creation of the process.
- Private
  - Processor register
  - Stack
- Shared
  - Memory
  - Resources (files, ...)

Process structure

- Threads require changes on processes
  - Each thread has a local TCB
    - Thread Control Block.
- Each thread contains its own stack
  - Local variables are local to each thread
- Threads share
  - Code
  - Global variables
  - Resources (FILES, IPC)



Threads inside processes



Shared	Not Shared
PID / PPID Terminal Open files Timers Resource limits	Thread ID Signal mask Thread specific Data Alternative signal stack Errno CPU affinity Stack

Thread API

- There are several APIs
  - Win32 threads.
  - C-Threads (user level)
  - Pthreads
    - POSIX IEEE 1003.1c, published in 1995

Simple Threads API

- **`void thread_create (thread, func, arg)`**
  - Create a new thread, storing information about it in thread.
  - Concurrently with the calling thread
    - thread executes the function func with the argument arg.
- **`void thread_yield ()`**
  - The calling thread voluntarily gives up the processor to let some other thread(s) run.
  - The scheduler can resume running the calling thread whenever it chooses to do so.

Simple Threads API

- **`int thread_join (thread)`**
  - Wait for thread to finish if it has not already done so;
    - then return the value passed to thread_exit by that thread.
  - thread_join may be called only once for each thread.
- **`void thread_exit(ret)`**
  - Finish the current thread.
  - Store the value ret in the current thread's data structure.
  - If another thread is already waiting in a call to thread_join, resume it.

POSIX Thread API

- POSIX defines functions for the management of threads
  - Functions/data started with the prefix pthread_
- Definitions available in the pthread.h file
- Code should be linked with the pthread library
  - -lpthread

POSIX Thread creation

- The main() method comprises
  - a single, default thread.
- **`pthread_create()`**
  - creates a new thread and makes it executable.
- The maximum number of threads that may be created by a process in implementation dependent.
- Once created, threads are peers, and may create other threads.

Thread creation

```
int pthread_create(
        pthread_t *thread, pthread_attr_t *attr,
        void *(*start_routine) (void *),  void *arg);
```

- 1st parameter - Pointer to thread identifier (out)
- 2nd parameter - Pointer to thread attributes (IN)
  - Can be NULL .
- 3rd parameter - Pointer to function containing the thread code
  - Function should be: void * (func*) (void * arg).
- 4th parameter - Pointer to thread arguments (IN)
  - Pointer to array, structure, int, ... (can be NULL)
- Returns 0 if successfulsdf
-

Thread termination

- Several ways to terminate a thread:
  - The thread function is complete and **returns**
  - The pthread_exit() method is called
- The **exit()** method is called and terminates all threads
- **void pthread_exit(void *retval);**
  - terminates calling thread
  - **Retval** points to value accessible using pthread_join

Thread termination

- If the main thread finishes with **pthread_exit**
  - the other threads will continue to exist
- The **pthread_exit** does not close files
  - any files opened inside the thread will remain open
- The **pthread_exit** does not free memory
  - All allocated memory remains

Thread joining

- A thread terminates itself by calling
  - **`int pthread_exit(void *ret)`** or **`return ret_val`**
  - The 1st parameter is a pointer to any data type
    - Memory location should be accessible outside (either global variable or malloc)
- After pthread_exit some resources are maintained
  - Until corresponding pthread_join().
- Any other thread can retrieve returned data
  - int pthread_join(pthread_t thread, void **retval);

Wait for a thread

- A thread waits for another thread executing pthread_join
  - To release resources
  - To fetch returned data
  - **int pthread_join(pthread_t thread, void **retval);**
    - 1st parameter - thread identifier.
    - 2nd parameter - Pointer to location of returned valued
- function waits for the thread specified by thread to terminate.
- If that thread has already terminated, then pthread_join() returns immediately.
- Only one thread can wait/join another thread

in/out Data transfer

- Data can be transmitted to the thread in several ways
  - Global variables
    - Accessible by all threads (synchronization should be applied)
  - 4th parameter of pthread_create
    - This parameter points to any data structure the programmer defines
    - Not use same memory location to multiple threads
      - Coherency not guaranteed
- Out data follows similar pattern

Data Transfer patterns

- Each thread need to do its work
  - Independent
  - Complementary
- Each thread requires
  - Shared data
  - Private data
  - Individual data
    - Supplied at creation time
    - 4th argument of pthread_create / function argument

Data transfer into threads

- Thread accepts a (void *)
  - void * - 8 bytes
  - int, float - 4 bytes
  - long, double – 8 bytes
  - Char – 1 byte

```
void * thread_func(void * arg){
    int val = (int) arg;
```

- pthread_create accepts scalars

```
for (int i = 0; i < 4; i++){
        pthread_create(&t_id[0],NULL,
                       thread_func, i);
 }
```

Data transfer into threads

- Thread accepts a (void *)
  - void * - 8 bytes
  - All pointers – 8 bytes

```
void * thread_func(void * arg){
    int val = *((int*) arg);
```

- pthread_create accepts scalars

```
for (int i = 0; i < 4; i++){
        pthread_create(&tid[0],NULL,
                       thread_func, &i);
 }
```

BAD! BAD! BAD!

Data transfer into threads

- Thread accepts a (void *)
  - void * - 8 bytes
  - All pointers – 8 bytes
- pthread_create accepts scalars

```
void * thread_func(void * arg){
    int val = *((int*) arg);
```

Data transfer into threads

```
for (int i = 0; i < 4; i++){
  int * int_p = malloc(sizeof(int));
  *int_p = i;
  pthread_create(&t_id[0], NULL, thread_func, int_p)
}
```

Data transfer out of threads

- Threads return a (void *)
  - Can return a scalar

```
void * thread_func(void * arg){
    int val = ............;
    return val;
}
```

```
    void * ptr_ret;
    pthread_join(t_id[0], &ptr_ret);
    printf("thread return %d\n", (int)ptr_ret);
```

Data transfer out of threads

- Threads return a (void *)
  - Can return a pointer

```
void * thread_func(void * arg){
    int val = ..........;
    return & val;
}
```

BAD! BAD! BAD!

```
    void * ptr_ret;
    pthread_join(t_id[0], &ptr_ret);
    printf("thread return %d\n", *(int*)ptr_ret);
```

Data transfer out of threads

```
void * thread_func(void * arg){
    int val = ............;
    int * ptr_ret = malloc(sizeof(int));
    *ptr_ret = val;
    return ptr_ret;
}
```

```
    void * ptr_ret;
    pthread_join(t_id[i], &ptr_ret);
    printf("thread return %u\n" * (int *)ptr_ret);
    free(ptr_ret);
```

Access to shared data....

- Concurrency
  - e.g. multiple threads increment same variable
- Random sleeps
- Small loops

```
void * thread_func(void * arg){
   for (int i = 0; i < 10000; i++){
      usleep(random()%500000);
      n++;
```


```
void * thread_func(void * arg){
   for (int i = 0; i < 10; i++){
    n++;
```

Access to shared data....

- Isolate each thread
  - Arrays or local variables
  - Processing after the join

```
void * thread_func(void * arg){
  int n_thread = (int)arg;
  n_array[n_thread] = 0;
  for (int i=0; i<LENGTH; i++)
     n_array[n_thread]++;
```

```
void * thread_func(void * arg){
  int n = 0;
  for (int i = 0; i < LENGTH; i++)
    n++;
  return n;
}
```